YADA: Yet Another Distributed Architecture for Real-Time Robotic Control Systems

Peter Godart, Peter Vieira, Gene Merewether, and Wyatt Ubellacker
NASA Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, Pasadena, CA, 91109, USA
{*Peter.T.Godart, Peter.Vieira, Gene.B.Merewether, Wyatt.L.Ubellacker*}*@jpl.nasa.gov*

***Abstract*— This paper presents YADA, a new software architecture for real-time robotic control systems that is minimal, modular, and fully transparent. YADA divides control software into decoupled behavior, user-interface, and hardware-level bus modules. This decoupling at the module level is accomplished by auto-generating human-readable message types that are tailored to the hardware topology of the current system. These message types provide modules with a common framework for exchanging state information and relaying commands to devices while being agnostic to the communication protocol itself. We also detail how to structure behavior and bus modules to facilitate modularity and flexibility with third party software. YADA has been used with success on several technology development testbeds at JPL, an example of which is given in this paper, and has proven to provide developers a light-weight and highly reconfigurable system for efficient debugging and practical code sharing.**

***Index Terms*— Real-time robotics, software architecture, hardware modularity, reconfigurable, distributed systems.**

I. Introduction

There is an ever-increasing trend in the robotics software community towards development using frameworks that abstract away the more utilitarian functions of inter-process communication, organization of state information, and the scheduling and execution of processes. In theory, such platforms as those surveyed in [1] free developers from the more mundane aspects of robotics software development, allowing them to focus on advancing the state of the art instead. We have found, however, that in practice many of these "middleware" software platforms abstract too much functionality from the user, inhibiting real-time control, complicating the usage of external libraries, and making debugging difficult by hiding code and functionality from the user. At JPL, we have the added complication that our software must ultimately run embedded on extraterrestrial spacecraft with limited memory and processing power. By designing and testing control algorithms using middleware with large and complex external software elements, we often have to start from scratch in transitioning to the flight system. Furthermore, many of our projects use multiple testbeds with the same core software system but require that we often swap out hardware and behavioral control modules for testing, a practice that is not easily accomplished with previous frameworks.

Many of these issues are solved by strict decoupling of software components. For example, a motor controller interface should merely execute the commands it receives and not depend on a robot's physical topology or high-level behavioral control algorithms. Architectures following this principle have become prevalent since they were first described in [2], and later at JPL in [3] and [4]. More modern implementations like RoboComp [5] provide additional tools for developing and analyzing discrete software modules, but they are either tied to some complicated, hidden middleware that makes debugging difficult, or they do not interface well with third-party software. Our goal with YADA was therefore to design a minimal, highly modular architecture that can leverage the large, active open-source communities backing external software like ROS [6]. We also required that hardware interface modules be decoupled from high-level control modules to facilitate rapid hardware prototyping while minimizing developer overhead. Finally, we aimed to implement an architecture in which all abstracted elements regarding communication between software components are still exposed to the user for ease of debugging.

II. YADA Overview

A. Definition of Terms

Throughout this paper, we use **module** to refer to a self-contained set of source code files that are compiled (if necessary) into a stand-alone communication node. All modules run synchronously at user-defined loop rates and communicate asynchronously with each other. **Behavior modules** receive and process state information and subsequently perform some function (e.g. path planning). **User interface (UI) modules** receive, process, and display state information, or they provide an interface for users to send commands directly to other modules. **Bus modules** communicate with hardware, execute device-level commands sent from behavior and UI modules, and send device state information back. Bus modules also contain configuration files that allow message types and helper functions to be auto-generated by **MotGen**, a YADA-specific auto-coder, further described in

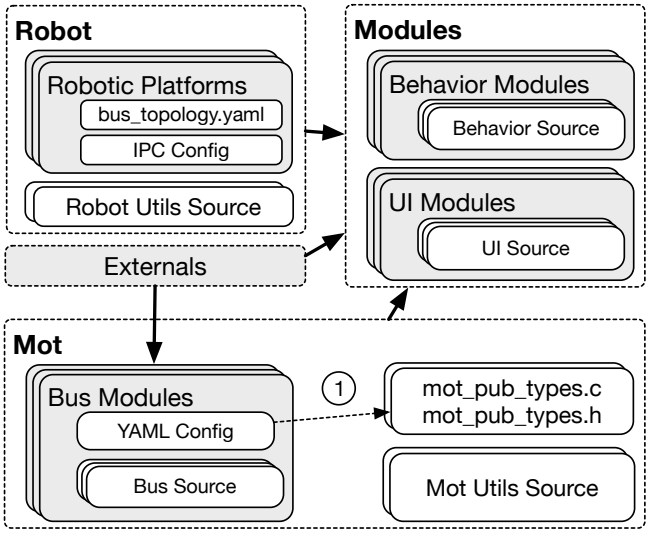

Fig. 1. High-level YADA organization. Solid lines show compilation dependencies between YADA components. Shaded boxes represent interchangeable elements. The dashed arrow labeled "1" indicates that mot_pub_types.c and mot_pub_types.h containing message structs are auto-generated from the YAML config files for each bus module.

Section III-B. Consequently, we break out bus modules and their associated auto-generated files into a separate software component called **mot** (pronounced as the first syllable of "motor"), nomenclature borrowed from JPL flight software. In the context of mot, we use **bus** to mean a communication bus that links a control computer with external hardware. We use **device** to mean a hardware and/or communication node on a particular bus.

B. High-Level Description

YADA is broken down into four components, *robot*, *modules*, *mot*, and *externals* that divide and organize configuration information and source code, as illustrated in Fig. 1. *Robot* contains one or more robot platform directories, each containing information defining their respective bus topologies and configuration parameters. *Modules* contains behavior and UI modules. *Mot* contains bus modules and their auto-generated message structs and helper functions, with which behavior, UI, and bus modules communicate with one another. Finally, *externals* contains pre-compiled libraries. As shown by the shaded regions in Fig. 1, most of the software base is interchangeable. Using version control software, swapping out modules and externals relevant to the current robot platform can be scripted, eliminating unnecessary code. Our recommended flexible build system, described in Section VI, enables seamless recompilation.

C. Inter-Module Communication

As an architecture, YADA emphasizes modularity at the *module* level. That is, users are able to easily switch out behavior, UI, and bus modules. Central to this effort is our framework for interprocess communication (IPC) between modules. YADA is agnostic to the particular flavor of IPC used; however, it is important that the publish/subscribe pattern, or some variant as summarized in [7], be used. YADA is definitively not "middleware", and thus should not require any intermediate message brokering. It is also crucial that modules be able to run in separate processes and on separate computers, as shown in Fig. 2. The publish/subscribe pattern is therefore a natural choice, providing modularity up to the computer level. Within the publish/subscribe pattern, YADA provides a system for auto-generating common message types that all modules use in communicating with one another. These message types, auto-generated depending on the hardware present on a robot platform, enable users to swap out bus modules with minimal code changes.

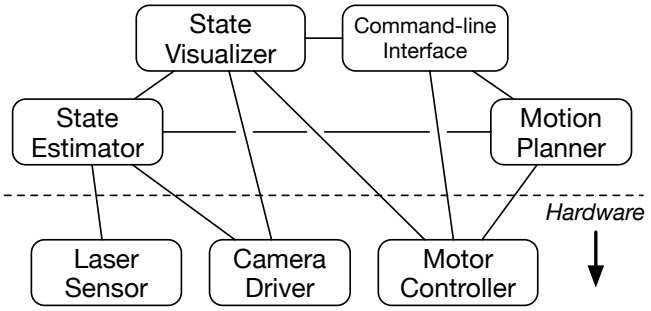

Fig. 2. Example YADA layout showing inter-module communication.

III. YADA Data Structures

A key feature that enables the decoupled modularity that YADA provides is the enforcement of common data structures between all modules. In our C implementation of YADA, we defined data structures for device configuration, device state, and bus-level commands. As illustrated in Fig. 3, we first define `mot_config_t`, a struct that contains a union of config structs, one for each device type used in the current robot platform. Each config struct contains variables that correspond to parameters relevant to only that device. Next we define `mot_state_t`, which contains a union of state structs, one for each device type used in the current robot platform. Like the config struct, the state struct contains state variables pertinent only to its specific device. Finally, we define a `mot_cmd_t` struct that contains a union of all command structs for each bus. We use a command type enum to dereference the appropriate union member. State structs are used to relay state information from bus modules to behavior and UI modules and command structs are used to send commands the opposite direction.

We use unions in our data structures so as to be able to store device information in a single array without wasting memory. To do this, we define another struct,

```
// Language: C

typedef struct {
  union {
    motor_controller_config_t
        motor_controller_config;
  };
} mot_config_t;

typedef struct {
  int msg_sender;
  union {
    motor_controller_state_t motor_controller_state;
  };
} mot_state_t;

typedef struct {
  bool execute;
  int msg_sender;
  mot_cmd_type_t type;
  union {
    move_position_cmd_t move_position_cmd;
  };
} mot_cmd_t;

typedef struct {
  mot_device_type_t type;

  mot_config_t config;
  mot_state_t state;
  mot_cmd_t cmd;
} mot_device_t;
```

Fig. 3. Fragment from MotGen output showing YADA data structures.

`mot_device_t`, that stores a single `mot_config_t`, `mot_state_t`, and `mot_cmd_t` struct for each device. As a result, we can fully describe a bus using an array of `mot_device_t` structs, where the position of a device in the array corresponds to the position of that device on its physical bus. To describe a complete robot platform, we define a two dimensional array of device structs, in which the first dimension refers to the specific bus and the second to the specific device. For convenience, we wrote robot utility functions to facilitate filling out and retrieving data from these arrays. For example, `robot_pub_get_device(bus,device_index)` returns a pointer to the `mot_device_t` struct at position `device_index` on bus `bus`.

A. Sending State and Command Messages

In communicating with behavior and UI modules, bus modules send an array of state structs (one for each device on its bus). The recipients use a mot utility function, `mot_pub_parse_state_msg()`, to copy the state information into the correct position in their robot array. This function uses the `msg_sender` struct member within `mot_state_t` to discern from which bus module the state information came. Similarly, to send commands directly to bus modules, behavior and UI modules use a mot utility function, `mot_pub_send_commands()`, to extract the relevant command structs from their robot array and send them as arrays to the appropriate bus module. To command a particular device, a module fills out the command variable fields in the correct position in the array and sets the `execute` boolean to true. The command recipient will then attempt to execute the command if that particular device is capable of executing it. For example, a sensor could conceivably be commanded to change its position, which would not make sense. It is therefore incumbent upon the bus modules to filter out such nonsensical requests.

B. MotGen Auto-Coder

A key attribute of YADA is that behavior and UI modules share a messaging framework for communicating with bus modules. To promote modularity, the message content is tailored precisely to the devices being used on a given robot platform. In our C implementation of YADA, our Python-based auto-coder, "MotGen", takes YAML configuration files from each bus module and adds members to the previously described unions within the `mot_config_t`, `mot_state_t`, and `mot_cmd_t` structs. For example, MotGen would take the YAML file shown in Fig. 4 and auto-generate the human-readable structs file shown in Fig. 3. Not shown in the latter figure are the definitions of the member structs in each union. These are defined earlier in the file and simply contain the `type` and `name` fields copied from the configuration file. The `motor_controller_config_t` struct would therefore contain `double position`, `double velocity`, and `double current` members. If a user wants to use a non-standard C type, they must add the appropriate header file under the `includes` map key, also shown in Fig. 4. Additionally, in Fig. 3, `mot_device_type_t` and `mot_cmd_type_t` are enums that are auto-populated by MotGen with all of the devices and command types respectively across all bus modules present in mot. The output of MotGen is fully transparent to the user, making system-level debugging much easier than it is using other software frameworks that hide messaging code.

IV. Specifying Hardware Topology

YADA places all relevant hardware topology information in one place. For each robot platform, a single YAML file includes which devices appear on which bus and in what order, as well as parameters such as an actuator's gear ratio. We chose YAML [8] due to its readability and open-source support for common programming languages. Our bus topology file, an example of which is shown in Fig. 5, uses YAML maps and sequences to encode information. Each bus type (e.g. RS-485) is specified as a map key whose value is a sequence of instantiations of that bus. In Fig. 5, for example, `rs485_bus` has one instantiation, named `limb_1`. Each element in this sequence is itself a map

```yaml
# Language: YAML 1.2

module_name: rs485_bus
includes:
  - rs485.h
state:
  - device_type: motor_controller
    state_variables:
      - name: position
        type: double
      - name: velocity
        type: double
      - name: current
        type: double
  # Additional device types here
config:
  - device_type: motor_controller
    config_variables:
      - name: units[256]
        type: char
      - name: gear_ratio
        type: double
      - name: max_speed
        type: double
  # Additional device types here
commands:
  - name: move_position
    args:
      - name: position
        type: float
      - name: velocity
        type: float
  # Additional commands here
```

Fig. 4. Example MotGen input for auto-generating message structs.

comprised of a name, optional bus-level details (e.g. baud rate) encoded as key/value pairs, and a required map key called `devices`. The value of `devices` must be a sequence of device parameter maps, whose order corresponds to their physical bus position.

At runtime, this YAML file is read in by each module that requires knowledge of device configuration using our robot utility functions that wrap an open-source YAML parser [9]. The parameters for each device are stored in a `mot_config_t` struct using separate helper functions defined in each bus module. Typically, only bus modules require specific details for the devices on their associated bus, while behavior and UI modules need only the high-level bus/device layout. If a behavior module needs to know more specific information about a device (e.g. actuator gear ratio), it can call a bus module's device parsing function to fill out their copy of that device's `mot_config_t` struct.

V. YADA Modules

A. Bus Modules

Bus modules are different from higher-level behavior and UI modules in two important ways. First, bus modules contain extra information necessary for MotGen to auto-generate message types for communication with higher-level

```yaml
# Language: YAML 1.2

rs485_bus:
  - bus_name: limb_1
    devices:
      - type: motor_controller
        name: wrist
        gear_ratio: 156
        max_speed: 6.2
      - type: motor_controller
        name: elbow
        gear_ratio: 310
        max_speed: 3.14
      # Additional devices on bus here
  # Additional buses of type rs485_bus here
```

Fig. 5. Example bus topology YAML file.

modules. Second, we often want to run multiple instances of the same bus module, as we often have multiple buses of the same type. For a robot with multiple arms, for example, it would be convenient to operate each arm on a separate bus, in case one were to crash, a system-level architecture described previously in [4]. For this reason, we also include an integer variable, `msg_sender`, in the `mot_state_t` struct as shown in Fig. 3. This variable allows other modules receiving an array of state structs to discern from which bus module that array came. This means that all modules must know about each bus module's unique identifier. Our implementation of YADA maps this unique identifier to the order in which a given bus module appears in the bus topology YAML file. While high level modules can read in this order directly, there is ambiguity at the bus module level, and thus we also pass in the unique identifier as an option to the bus module executable.

B. Behavior Modules

Behavior modules control bus modules to perform some set of actions. Due to the similarity between different behavior modules, a python auto-coder was created that generates a new behavior module from a template, naming all of the files and variables according the desired name of the module. A key aspect of this auto-coder is that it contains a *core* and a *project* directory, allowing core functionality to be shared across robot platforms or projects, while project functionality can be customized for each robot platform. This produces several advantages: 1) it forces standardization, which reduces bugs across code, removes the need to reinvent the wheel, and makes it easier for people to understand the code, 2) it allows developers to focus on algorithm development instead of boilerplate code, and 3) enables more code reuse due to the core/project paradigm. This paradigm consists of having two independent directories, core and project, that depend on each other. The core contains generic command definitions and behaviors, while the project contains project-specific commands and behaviors. For example, an Arm

module might contain all the basic motion algorithms in the core, such as joint and cartesian space trajectories and path planning, while the project could add commands and behaviors for controlling a specific gripper mechanism or using a platform-specific kinematic solver. We chose to implement behaviors using Hierarchical State Machines (HSMs) (e.g. [10], [4], and [3]) but any other scheme could be substituted in to achieve the same result. Our behavior module template is set up to receive messages from the command-line interface and communicate with bus modules. Adding a behavior module to a robot platform involves adding it to the modules directory, the robot platform CMakeLists.txt file, and the Cmd and IPC files.

VI. Build System

For our C implementation of YADA in Ubuntu 14.04, we use CMake to aid in the compilation process due to its ease of use, flexibility, and proven reliability [11]. Each behavior, UI, and bus module has its own CMakeLists.txt configuration file. Users can specify which of these modules' configuration files gets included with a CMakeLists.txt file located within the current robot platform's directory. In this way, directories and references can be different for each robot platform in a particular YADA instantiation, facilitating modularity at the build level and consequently code reuse. At the configuration step, an option is passed into the `cmake` binary that specifies which robot platform to use for compilation. A CMakeLists.txt file within the YADA root directory uses this option to include the appropriate platform's CMakeLists.txt.

The YADA root CMakeLists.txt file also includes configuration files for mot and optionally external libraries. The mot CMakeLists.txt file is responsible for calling the MotGen binary that auto-generates the message structs header and associated helper functions files in that same mot directory. These files are then compiled into the mot library, against which all modules link. To actually compile YADA, after running `cmake`, the user simply runs GNU `make` to compile all executables and libraries. Note that for organizational purposes, an out-of-source build was chosen, which results in all build files being put in a separate "build" directory and all module binaries being put in a separate "bin" directory at the YADA root directory. Fig. 7 shows an example of how this build process is structured. In this example, the user would run `cmake -Drobot="Spacecraft Emulator"` to set up the appropriate build and configuration files, and then `make` to compile module binaries and the mot library.

VII. Comet Sampler Application Example

An example of YADA being used on a real system is JPL's comet sampler testbed. Here JPL is exploring a notional mission concept in which a spacecraft would fly to a comet and retrieve a sample using an on-board sampling mechanism [12]. For research, an electro-mechanical system has been developed to emulate a 2000 kg spacecraft with a comet sampling mechanism. Fig. 6 shows this system, which consists of a 2000 kg cubic structure with seven compressed air tanks used to control air bearings and thrusters, as well as a single-actuator sampling mechanism. To emulate the sampling event, the system floats on air bearings, controls its thrusters to approach the comet simulant based on position data from a motion tracker, fires the sampling mechanism, and retreats from the simulant. This software/hardware system is comprised of an operator control station, the untethered spacecraft, a spacecraft computer, an Arduino Due controlling the air bearings and thrusters, a servo drive controlling the sampling mechanism actuator, and a motion tracker tracking the position of the spacecraft [13]. Fig. 7 shows the YADA layout for this system.

Fig. 6. Actual Comet Sampling Spacecraft Emulator hardware.

A similar system, comprised of the sampling mechanism at the end of a robotic arm, is also used in testing. The same YADA system can be used in controlling this testbed simply by changing the contents of the bus topology file and including the appropriate bus and behavior modules. We found that using YADA in switching between these two testbeds greatly reduced the time required to both set up basic functionality and develop more complex control algorithms due to modularity and ease of code sharing.

VIII. Discussion

A. Replacing Bus Modules

Replacing YADA bus modules requires minimal effort. For example, if a user wishes to replace a particular motor controller device that uses a different bus communication protocol, they would first edit the bus topology YAML file to reflect the changed bus and device type. Next, because our implementation is written in C, the user would have to edit how the `mot_device_t` struct's unions are dereferenced within behavior and UI modules. While this seems cumbersome, we actually found that the explicit reference to which bus modules are being used reduced bugs in logic and

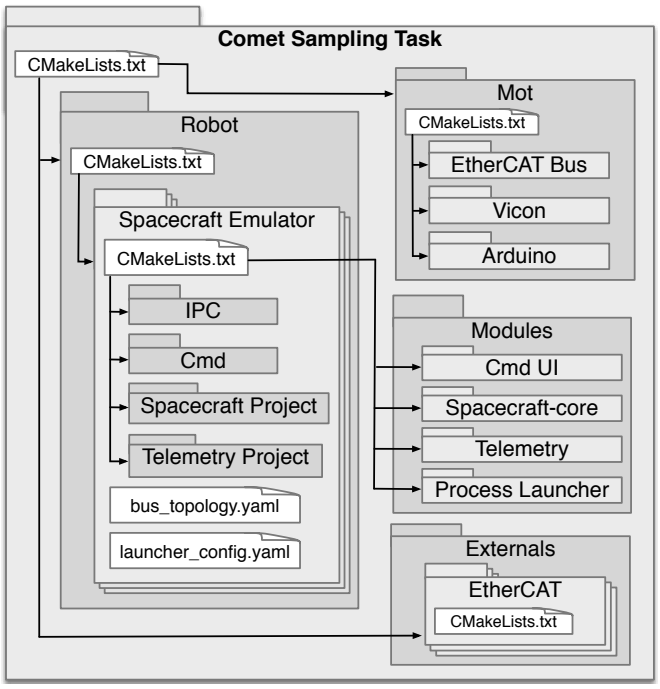

Fig. 7. Comet Sampling Spacecraft Emulator build system. Not shown are CMakeLists.txt files in each of the modules within the "Modules" directory.

made code more readable. Further, as previously described, behavior modules are set up such that core functionality is retained while project functionality is tailored to the particular bus topology. Therefore, each bus topology has its own set of project files and so the user does not have to keep replacing code every time they swap out a bus module.

B. Compatibility With SysML Auto-Coder

As discussed in [14], outlining robotics control software using block diagrams can be beneficial. Generating state machines and hardware topologies is well-suited to a block diagram format and can be developed with a tool such as MagicDraw [15]. YADA interfaces with this workflow with no modification by providing standard formats for the HSM and bus topology files, as well as a library of functions and devices types available for use in the block diagrams. Though the implementation outlined in [14] is done with MagicDraw, YADA can extend to any number of visual diagramming tools provided they conform to the YADA interface. Once the user draws block representations of state machines and hardware layout, they are converted into code and configuration files respectively following the techniques described in Section III. This process can quickly be iterated for any change or additions to the behaviors and devices.

IX. Summary

In summary, YADA is not middleware, but rather a minimal software architecture and method for auto-generating transparent message types and helper functions, which facilitate interchangeable software elements. This auto-generated code provides modules with a common language for exchanging hardware-level state information and commands and is precisely tailored to the hardware configuration of the current robot platform. A robot platform's hardware topology and configuration is expressed using a single human-readable YAML file that is parsed by all modules at runtime. Our suggested flexible build system facilitates switching between robot platforms with minimal effort. As a result of these features, YADA significantly reduces developer overhead, especially in technology development systems that require rapid prototyping of different hardware configurations and behavior control modules. Furthermore, its transparent auto-coding scheme makes debugging simple and facilitates practical code reuse since there are no hard-coded hidden module dependencies.

Acknowledgment

The research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (NASA). Copyright 2017 California Institute of Technology. U.S. Government sponsorship acknowledged.

References

[1] A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *Journal of Robotics*, vol. 2012, 2012.

[2] R. Brooks, "A robust layered control system for a mobile robot," *IEEE journal on robotics and automation*, vol. 2, no. 1, pp. 14–23, 1986.

[3] K. Edelberg *et al.*, "Autonomous localization and acquisition of a sample tube for mars sample return," in *AIAA SPACE 2015 Conference and Exposition*, 2015, p. 4483.

[4] S. Karumanchi *et al.*, "Team robosimian: Semi-autonomous mobile manipulation at the 2015 darpa robotics challenge finals," *Journal of Field Robotics*, vol. 34, no. 2, pp. 305–332, 2017. [Online]. Available: http://dx.doi.org/10.1002/rob.21676

[5] L. Manso *et al.*, "Robocomp: a tool-based robotics framework," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2010, pp. 251–262.

[6] M. Quigley *et al.*, "ROS: an open-source Robot Operating System'," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, 2009, p. 5.

[7] P. T. Eugster *et al.*, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[8] "YAML Ain't Markup Language (YAML™) Version 1.2," http://www.yaml.org/spec/1.2/spec.html, accessed: 2016-09-30.

[9] "yaml-cpp," https://github.com/jbeder/yaml-cpp, 2015.

[10] M. Samek, *Practical statecharts in C/C++: Quantum programming for embedded systems*. CRC Press, 2002.

[11] K. Martin and B. Hoffman, *Mastering CMake*. Kitware, 2010.

[12] P. Backes *et al.*, "Biblade sampling tool validation for comet surface environments," in *Aerospace Conference*. IEEE, 2017.

[13] S. Monk, *Programming Arduino Next Steps: Going Further with Sketches*. McGraw Hill Professional, 2013.

[14] P. Godart, J. Gross, R. Mukherjee, and W. Ubellacker, "Generating real-time robotics control software from sysml," in *Aerospace Conference, 2017 IEEE*. IEEE, 2017, pp. 1–11.

[15] NoMagic, *MagicDraw*. http://www.nomagic.com/products/magicdraw.html, visited 10/4/2016, 2016.